TRANSITION AIDS
SYSTEM V

unix* system

*Trademark of Bell Laboratories

301-908

FAUST

UNIX™ System

Transition Aids

System V

January 1983

© 1983 Western Electric Company, Incorporated

PDP, VAX, DEC, UNIBUS, MASSBUS,
and SBI are trademarks of Digital Equipment Corporation.

UNIX is a trademark of Bell Laboratories

TABLE OF CONTENTS

TAB	TITLE	PAGES
A	UNIX Operating System Environment Changes: System III to System V	5
B	Converting to a 1K-block File System	3
C	Changes to the UNIX System Archive Format	4
D	Common Object File Format	22
E	User Level Changes to the C Language	6

UNIX Operating System Environment Changes:

System III to System V

UNIX† Operating System Environment Changes: System III to System V

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

There have been many additions and modifications to the UNIX Operating System for the System V release. Generally, additions are always welcome to any effort. However, some changes require the administrator or user to change files or procedures used in the normal operation or configuration of the system. This document highlights those features in System V which might *change* the operation of the system. It does not include all the features in System V. For a complete list of features and enhancements you should consult the *System V Prospectus*.

2. Device Drivers

2.1 General Disk Driver

The general disk driver supports several different standard disks (RP04/5/6, RP07, RM80, RM05) on a single controller and will also support up to 4 controllers. The *gd*(7) driver is available for both the DEC* VAX* and PDP*-11 lines. These devices are specified by the label `disk` in the configuration file.

2.2 General Tape Driver

The general tape driver supports the TE16/TM03, TU77/TM03, and TU78/TM78 MASSBUS[1] tape drive/controller pairs. The new *gt*(7) driver replaces the older *ht*(4) driver for the TE16/TM03 tape drive/controller on the VAX line only. These devices are specified by the label `tu1678` in the configuration file.

2.3 KMC/DZ Asynchronous Line Interface

The KMC11-A is no longer supported as the UNIBUS[2] controller. This controller only had 1K byte of memory and was not suitable for handling both input and output. The System III version of the DZ/KMC driver only used the KMC for output to the DZ. The KMC11-B is fully supported for DZ assist, and the interface is now much more efficient in passing characters back to the cpu (input). The DZ devices are specified by the label `dzb` in the configuration file.

2.4 Bdevsw, Cdevsw, Line Disciplines, etc...

For those interested in moving device drivers from System III to System V, note that there have been changes to the file `/usr/include/sys/conf.h`. The following information is not comprehensive enough to make the required changes, a detailed look into the operation of the drivers is required.

The pointer to the system I/O buffers has been removed from `bdevsw` because the buffers are hashed by device number. In a separate change, a pointer to a new routine is now present, `d_print` to provide a uniform error message printing routine. The pointer to the `struct tty` in `cdevsw` and the new `struct termsw` were added to handle multiple line disciplines.

---

† UNIX is a trademark of Bell Laboratories.

* DEC, VAX and PDP are trademarks of Digital Equipment Corporation.

1. MASSBUS is a trademark of Digital Equipment Corporation.
2. UNIBUS is a trademark of Digital Equipment Corporation.

3. File System

The file system structure has been enhanced to increase performance but maintains compatibility with the System III file system. The new file systems use a 1K byte block size, but the older System III, 512 bytes per block file systems may still be mounted and used, as is. The new 1K byte block size reduces the number and time of accesses to the disk. The system buffers have been expanded to support the 1K byte block size. The operating system distinguishes between the two types of file system and handles them with no discernible difference to the user other than performance.

The key to the difference in format is contained in `struct filsys` in `/usr/include/sys/filsys.h` which describes the *super-block*. The *boot-block* and the *super-block* are in the same place and are the same size for both file system structures. The *boot-block* is the first 512 bytes of a file system and the *super-block* is the next 512 bytes. In `struct filsys` there are two elements, `s_magic` and `s_type`, used for file system size identification. New file systems created by /etc/mkfs (for 1K byte blocks) or /etc/omkfs (for 512 byte blocks) have `s_magic` set to `FsMAGIC`. For 1K byte file systems the element `s_type` will be set to `Fs2b` and for 512 byte file systems the element `s_type` will be set to `Fs1b`. If `s_magic` is not set to `FsMAGIC` the file system is assumed to be of the "old" 512 byte block variety.

Except for the *boot* and *super* blocks (blocks numbered 0 and 1), all other blocks in the file system are found at a byte offset equal to their block number multiplied by the size, in bytes, of the logical file system block.

There are very few commands which are affected by the difference in file system block size. These commands are usually involved in generating or reporting the sizes of file systems (e.g. *mkfs*(1M), *fsck*(1M), *df*(1M)). For compatibility these commands still report sizes in 512 byte blocks. In general there is no reason for a user to be concerned with the type of file system being dealt with. The labelit command (see *volcopy*(1M)) now reports the file system block size.

3.1 Initial Boot

System V will be delivered with 1K byte block root (/) and user (`/usr`) file systems. After the new system is read in and booted, the system administrator may copy local files from the site backups. Any of the standard UNIX system tools may be used to transfer the old files to the new file system. Care should be taken not to overwrite the new file system check program *fsck*(1M) or any of the accounting programs. In general it is wise not to overwrite any of the files delivered with the new system (except those intended for local modification).

3.2 Performance vs. Space

To choose whether to keep an old file system with 512 byte blocks or generate a new 1K byte file system, trade-off decisions must be made. The larger size file system provides better performance than the older file system because of the reduction of the number of disk seeks per file read. In addition, the buffers in System V have all been increased to support the larger file system so that use of a small block file system means that half of the system buffer space will be wasted for those blocks that are buffered (one buffer block is used per file system block regardless of the size of the file system block). On the other hand, a file system with a large number of small files (< 512 bytes) will have more disk space wasted if converted to the large block file system. In fact, small block file systems that are almost full may not fit on a large block file system using the same disk space. A transition tool *fsba*(1M) is supplied with System V to assist the administrator. *Fsba* measures the size of a 512 byte file system and estimates the number of physical (512 byte) blocks and the number of inodes required to make a 1K byte/block file system. Documentation accompanies this program explaining its function and use more thoroughly. The performance vs. space decision must be made by the system administrator before converting file systems.

3.3 Conversion of Existing File Systems

Any of the existing file systems may be converted to the larger block size. Since existing file systems are supported by System V, no special conversion tools are necessary. Files resident under an old file system may be copied to a new (1K byte block) file system using *cpio*(1) and *find*(1) or the *finc*(1M) and *frec*(1M) routines. Do not use *volcopy*(1M), as this command will restore a complete file system, including the small block structure.

3.3.1 Cautions Because of the possibility of additional wasted space on 1K byte block file systems, conversion of full and nearly full 512 byte block file systems may not be possible. Refer to the program *fsba*(1M) (file system block analyzer) to supply you with a summary of the space usage on a small block file system and a projection of the usage on a large block file system.

Also, because *cpio*(1) has a limit to the number of unique linked files it can remember, care must be taken when converting a file system with a large number of these files.

3.4 Re-Compilation of Users' Software

User code and application code that uses the standard I/O library should be recompiled to take advantage of the larger buffer size of the new library. The increase in size of the file system block and the standard I/O buffer are independent of each other in an operational sense. The recompiled code is fully compatible with both file systems and will be more efficient even when using 512 byte file systems.

3.5 Modification of Users' Software

User and application code not using standard I/O or doing its own buffering may need to be changed to take advantage of the new file system. Internal buffers used for I/O should be set to be a multiple of 1024 bytes so that they will efficiently match the internal buffer size of System V. The buffer size may be specified (in a portable way) by using the define constant `BUFSIZ` from the header file `/usr/include/stdio.h`.

User code and application code that accesses file systems directly must be examined (and probably changed) and recompiled so that they are in accord with the new file system structure.

4. Init & Getty

4.1 Init

The entries in the /etc/inittab (see *inittab*(5)) file have been changed as part of the new interface, so that it is not possible to use the old /etc/inittab file. More states are now possible for spawning programs, and users should read the manual page carefully to select the proper *action* when a process terminates. Changing from multi-user state to single-user state is done by using *init* as a command, however; the single-user state is now called s (instead of 1) so that the command

```
init s
or
init S
```

brings the system into single user mode. When changes are made to /etc/inittab, the administrator now uses the command

```
init q
```

to signal the init process of the change. The shell script *shutdown*(1M) is provided for the administrator to bring the system down to the single-user state gracefully. A command file /etc/brc (see *brc*(1M)) is now executed after the system is booted so that commands to download devices should be placed here. A similar file /etc/powerfail (see *brc*(1M)) is available for power fail. The sections for downloading device controllers (e.g. KMC11-B) and performing power failure recovery formerly appeared in the /etc/rc file.

In System III the file /etc/rc was called with three arguments so that the current state of the machine was known. The three arguments passed to /etc/rc in System III, the current state, the number of times in that state (previously), and the previous state are no longer passed directly to /etc/rc. This information is now generally available as the seventh, eighth and ninth fields of the command **who -r**. The following lines can be inserted at the beginning of the /etc/rc file so the positional parameters are at the same location as they are in System III.

```
set `who -r`
shift 6
```

Note, when checking for single-user state the character 'S' should be used, not '1'. For example, later in the same file one might have:

```
if [ ($1 = 2) -a ($2 = 0) ]
then
# commands for first time in multi-user state
fi
```

This will execute commands the first time the system is in state 2 (generally multi-user mode) after a reboot.

4.2 Getty

The line speed table is now in the file /etc/gettydefs, so that the sequence of line speeds can be edited online. (No recompilation of the *getty*(1M) command is necessary.)

The /etc/gettydefs file can be checked using the '-c' option of the getty command. The options to getty are slightly different and are described in the manual pages.

4.3 Related Changes

Because of the changes to *init*, the format and location of the /etc/utmp and /etc/wtmp files are not compatible with previous systems. Users who have written private copies of commands that use these files should be prepared to adopt the new format described in *utmp*(4). Library routines (*getut*(3c)) are provided to ease the conversion.

5. PDP-11 Overlay Loader

This loader was specifically modified for the PDP-11 operating system (OS). It is called by the *makefile* for the operating system and permits the use of an OS which is larger than the amount of memory that can be handled by the kernel memory management registers. This is not a true overlay loader in that there must be enough physical memory for the entire OS. Only the *text* portion of the OS is managed in this way, so that a limitation of 56K bytes must still be observed for the *data* space. The loader and associated software manipulate the memory management registers to access a larger amount of memory than is normally available to the kernel instruction space. The loader is given the names of the routines which will be 'overlay loaded', generally these are device drivers.

A previously unused byte is used in the object file header to store the increased size. There were several commands (e.g. size, sum) that were changed to recognize this size change.

6. Configuration

There are several additional parameters that must be specified and assigned values when making a new OS. The parameters are explained in the *Setting Up Unix* document.

6.1 Hashbuf

This parameter specifies the number of hash buckets to allocate for the hashing of the system buffers.

6.2 Iblocks (PDP-11 only)

The *inode data block address cache* was implemented on the PDP-11 to conserve data space by permitting the administrator to allocate the data block pointers separately from the *inodes*. An *iblock* is needed for each regular, directory, and fifo file that is open. Block and character special files do not need *iblocks*.

Converting to a 1K-block File System

Converting to a 1K-block File System

1. INTRODUCTION

This document describes how to convert to a 1K (1024) byte block file system from the standard 512-byte block file system. Problems that should be anticipated are discussed and how to recover or avoid them. The document also presents the tradeoffs involved when converting to a 1K-block file system. Finally, the (*fsba*(1M)) command (file system block analyzer) is presented as a tool to estimate the increase in number of blocks after file system conversion.

2. TRADEOFFS

The larger block (1024 byte) file system is intended to exploit higher system throughput by increasing the number of bytes transferred during each I/O operation and reducing the number of seeks necessary to read a file. Unfortunately, this higher throughput usually comes at the expense of additional space required on a file system.

It should be emphasized that the operating system does I/O transfers to/from disk in 1024 byte chunks under the new scheme. The internal system buffers are 1024 bytes. Fewer I/O transfers are needed to read the same file in a 1K-byte block file system versus a 512-byte block file system.

I/O transfers for a 512-byte block file system are 512 bytes, using only half of each system buffer, thereby wasting part of the system buffer.

Converting to a 1K-block file system could reduce the available disk space. Measurements indicate that for many file systems, the increase is in the range of 5-10%; however, some file systems may have more wastage because of their structure. For small files (size <= 512), space is wasted because the minimum block size is 1024 bytes. For large files, the indirect blocks are also double size so there is additional space dedicated to indirect blocks. Determining the amount of extra space is not straight-forward so the *fsba* command is supplied with the release to do the computation (see following sections).

3. HOW TO CONVERT TO A 1K BYTE BLOCK FILE SYSTEM

3.1 Using Two Disk Drives

- *fsba* /dev/rdsk13 ...

  Use the *fsba* tool to predict the additional number of blocks the file system will require after conversion. (Assume the character special device name of the file system we want to convert is **/dev/rdsk13**. Block special device names may also be used).

- *mkfs* /dev/rdsk23 blocks:inodes

  Create a new 1K-byte block file system with appropriate block and inode numbers using the *mkfs*(1M) command. (Assume the new file system is **/dev/rdsk23**.)

- *labelit* /dev/rdsk23 fsname volume

  Label the new file system with the same file system name as the original 512-byte block file system.

- *mount* /dev/rdsk23 /to-directory

  Mount the new file system in directory /to-directory.

- *cd* /from-directory

  Move to the root of the file system to be converted, in this case, /from-directory.

- *find* . -print | *cpio* -pdm /to-directory

  Copy every file in the original file system to the new file system using *find*(1) and *cpio*(1).

3.2 Using The Raw Tape

If a spare disk drive is not available, the raw tape could be used to convert the 512 byte file system as follows:

- *fsba* /dev/rdsk3

  Use the *fsba* tool to predict the appropriate number of blocks the file system will require after conversion. (Assume the character special device name of the file system to be converted is **/dev/rdsk3**.)

- *cd* /from-directory

  Move to the root of the file system to be converted.

- *find* . -print | *cpio* -oB > /dev/rtp0

  Copy every file in the file system to be converted to the raw tape using *cpio*(1). (Assume the raw tape file name is **/dev/rtp0**.)

- *cd* /

  Move to the root of the entire file structure.

- *umount* /dev/rdsk3

  Umount the file system to be converted.

- *mkfs* /dev/rdsk3 blocks:inodes

  Create a new 1k byte block file system in the same disk slice.

- *labelit* /dev/rdsk3 fsname volume

  Label the new file system with the same file system name as the original one.

- *mount* /dev/rdsk3 /from-directory

  Mount the new file system in directory /from-directory.

- *cd* /from-directory

  Move to the root of the new file system.

- *cpio* -iBdm < /dev/rtp0

  Copy the file system from the tape to the new 1k file system using *cpio*(1).

4. PROBLEMS

When using *fsba*, make sure the file system is either 'umounted' or mounted read-only to assure accurate results.

After conversion, system administrators should be aware that in some cases, the total number of blocks in the converted file system could potentially decrease. Consider, for example, a directory using more blocks than what its contents require (i.e., entries have been removed). After conversion, this directory **may** require less blocks.

5. RECOVERY

If the file system runs out of space when converting to the larger block size, one way to recover is to reduce the number of free i-nodes so there is more space available for data blocks. Alternatively, some of the files can be removed from the file system or a larger disk slice can be used.

6. FSBA (File System Block Analyzer)

Fsba(1M) determines the number of extra sectors needed when the file system logical block size is increased from 512 bytes to 1024 bytes.

Fsba determines how many disk sectors are currently allocated for the 512-byte block file system and how many disk sectors will be required for the 1024-byte block converted file system. *Fsba* also prints the number of allocated and free i-nodes for each file system.

If *fsba* reports a negative number as the total free sector value for the 1K-block file system, then the file system after conversion would be too big for that particular volume/file system size.

Fsba is available only on a VAX machine.

Usage example:

```
fsba /dev/rdsk13

 file-sys name :        ud   (/dev/rdsk13)
 total inodes :        32808
 allocated inodes :     14747
 free inodes :         18061

    ----------------------------------

 for old file system (512bytes/block)
 allocated sectors :    120403
 free sectors :        6746

    ----------------------------------

 for new file system (1024 bytes/block)
 allocated sectors :    129766
 free sectors :        -2617
 the entire volume size is not big enough

    ----------------------------------

 extra sectors needed for new file system :
                    9363   ( 7.8%)
 * 1 sector contains 512 bytes
```

This file system would not be able to fit in the original space allocated for it. In some cases, it is possible to provide enough free space in the converted file system by reducing the number of i-nodes. But here, the only way to convert is to either increase the file system size or to delete some files in the old file system.

Another example is:

```
fsba /dev/rdsk15
```

```
file-sys name :        sccs   (/dev/rdsk15)
total inodes :        14936
allocated inodes :     4483
free inodes :         10453

    ----------------------------------

for old file system (512 bytes/block)
 allocated sectors :    46814
 free sectors :        11091

    ----------------------------------

 for new file system (1024bytes/block)
 allocated sectors :    49274
 free sectors :         8631

    ----------------------------------

 extra sectors needed for new file system :
                    2460   ( 5.3%)
 * 1 sector contains 512 bytes
```

This case would fit but an additional 5% of space will be used.

7. CONCLUSION

The decision to convert an existing 512-byte block file system should be based on the distribution of file sizes, frequency of use and performance gains. Tradeoffs should be taken into consideration. On the average, converting will be worth the effort.

Fsba can be used to predict the increase in the number of disk sectors when converting to a 1K-byte block file system from the 512-byte block file system.

Changes to the UNIX System Archive Format

Changes to the UNIX System Archive Format

Bell Laboratories

1. Introduction.

This document describes the latest changes made to the archive file format used on the UNIX* operating system. This archive format was introduced to the VAX+-11/780 and 11/750 versions of the UNIX operating system as of System V. This document addresses only the archive formats on these processors.

The archive command *ar* is used to combine several files into one file. No restriction is placed on the contents of the individual files. Archives are used mainly as libraries to be searched by the link editor during the compilation process. This new archive format allows the link editor to search libraries more efficiently due to an initial symbol directory that acts like a table of contents for all members of the archive. Figure 1 shows the overall structure.

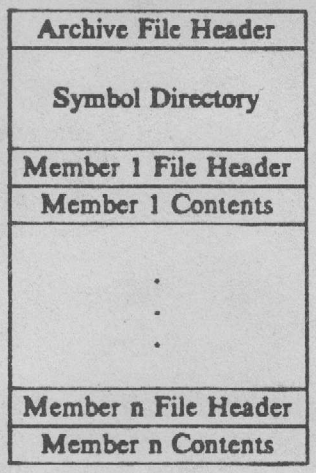


Figure 1. Archive File Format

The following sections of this document discuss each portion of the above format in some detail.

2. Archive Header

The archive header allow the link editor to identify this file as being an archive. Each archive begins with such a header which is made up of the components listed in table 1.

---

* UNIX is a trademark of Bell Telephone Laboratories, Inc.

\+ VAX and PDP are trademarks of Digital Equipment Corporation.

Bytes	Declaration	Name	Description
00-03	char	ar_magic	Magic number
04-19	char	ar_name	Name of archive
20-23	char	ar_date	Date of last modification
24-27	char	ar_syms	Number of symbol entries

TABLE 1. Archive Header Format

The archive magic number is defined to be the string <ar>. The *ar_date* field indicates the last time a change occurred anywhere in the archive, such as modifying a member, adding a member or deleting a member. The number of symbol entries in the table of contents will be zero if the archive does not contain object modules, that is, it is not being used as a library.

2.1 C Data Structure

The C data structure used for the archive header is as follows:

```
#define ARMAG      "<ar>"
#define SARMAG     4

struct  ar_hdr {                        /* archive header */
        char    ar_magic[SARMAG];       /* magic number */
        char    ar_name[16];            /* archive name */
        char    ar_date[4];             /* date of last modification */
        char    ar_syms[4];             /* number of ar_sym entries */
};
```

3. Symbol Directory

Each archive which contains members that are in Common Object File format (see UNIX system Reference manual *a.out*(4)) includes an archive symbol directory. This directory is created from the individual file member symbol tables. It is used by the UNIX link editor to determine which archive members must be loaded during the link edit process. Previously, the link editor would need to make one complete pass over the library reading each member to see if it needed to be loaded. The use of a symbol directory allows the link editor to perform random access on the library so it needs only to read the member entries that must be loaded.

The symbol directory immediately follows the archive header described above. The number of symbol directory entries is indicated in the archive header *ar_syms* variable. Each symbol directory entry has the format found in table 2.

Bytes	Declaration	Name	Description
00-07	char	sym_name	Symbol name used by link editor
08-11	char	sym_ptr	Location of symbol in archive

TABLE 2. Symbol Directory Format

The archive symbol directory is automatically created and/or updated by the UNIX system *ar* command.

3.1 C Data Structure

The C structure format used for a symbol entry is:

```
struct  ar_sym {                    /* archive symbol table entry */
        char    sym_name[8];        /* symbol name, recognized by ld */
        char    sym_ptr[4];         /* archive position of symbol */
};
```

4. Member Header

Following the archive header and symbol directory are the archive file members. Each file member consists of a file member header followed by the file member contents. The format of the file member header is given in Table 3.

Bytes	Declaration	Name	Description
00-15	char	arf_name	Name of file member
16-19	char	arf_date	Member last modification date
20-23	char	arf_uid	Member user identification
24-27	char	arf_gid	Member group identification
28-31	char	arf_mode	Member mode
32-35	char	arf_size	Size of file member in bytes

TABLE 3. Member Header Format

The user and group identification values and the mode value are the corresponding UNIX file system values for each member file.

Note that there is no provision for leaving empty areas within an archive.

4.1 C Data Structure

The C structure used for file member headers is:

```
struct  arf_hdr {                   /* archive file member header */
        char    arf_name[16];       /* member name */
        char    arf_date[4];        /* member date */
        char    arf_uid[4];         /* member user identification */
        char    arf_gid[4];         /* member group identification */
        char    arf_mode[4];        /* member mode */
        char    arf_size[4];        /* member size */
};
```

5. General Details

5.1 Portability

All information in the archive header, symbol directory and file member headers is stored in a portable format. All character data (e.g. archive/file member names) is inherently machine independent. All numeric data contained in the headers is stored as a four byte quantity. Special conversion routines exist for accessing this information. These routines are available in the UNIX

system *libld.a* library and a description of them can be found in the UNIX Reference manual page *sputl*(3X).

5.2 Transportability

Common format archives can be moved between UNIX systems as long as the portable archive command *ar(1)* is used. Conversion tools exist allowing archives created on UNIX systems not supporting this format to be converted into this format. A straightforward approach is simply to re-archive the files using the portable archive command. If this is not possible then the UNIX system command *convert*(1) can be used to convert pre-System V VAX archives into this format. Also, the UNIX system command *arcv* exists for the conversion of PDP-11/70 archives into this format. See Section 1 of the UNIX Reference manual for the exact usage of these commands.

5.3 Efficiency

For accessing and efficiency reasons each archive file member is forced to begin on a word boundary. To accomplish this null bytes may be inserted between files if necessary. However, the size of the member given in the file member header reflects the actual size of the file exclusive of padding.

Common Object File Format

Common Object File Format

Bell Laboratories

1. Introduction

The output file produced by the UNIX* system *as* assembler and the *ld* link editor is in a format called the *Common Object File Format* (COFF). Several target processors use this format and operating systems besides the UNIX system also use it. Hence the word *Common* is both descriptive and widely recognized as a unique name. Because systems other than the UNIX operating system use COFF, the format includes some symbols and fields that appear to be extraneous. These items are extraneous for UNIX operating system users, and are included only to maintain commonality.

Currently, this object file format is used for the

VAX+-11/780 and 11/750 UNIX operating systems.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file consists of a file header, optional header information, a table of section headers, the data corresponding to the section headers, relocation information, line numbers, and, finally, a symbol table. Figure 1 shows the overall structure.

The last three sections (relocation, line numbers, and the symbol table) may be missing if the program was linked with the -s option of the UNIX link editor or if the relocation information and symbol table were removed by the UNIX strip command. Also, if there were no unresolved external references after linking then the relocation information is no longer needed and will be absent.

An object file that contains no errors or unresolved references can be executed on the UNIX operating system.

The Common Object File Format is not only simple enough to be easily incorporated into existing projects, but advanced enough to meet the needs of yet unspecified operating systems. Some key features are:

- Applications may add system-dependent information to the object file without causing access utilities to become obsolete.
- A wealth of symbolic information is provided for the use of debuggers and other applications.
- Many modifications in object file construction may be made by the user or application at compile time.

2. Definitions

The object file specification uses the following terms:

---

* UNIX is a trademark of Bell Telephone Laboratories, Inc.

\+ VAX is a trademark of the Digital Equipment Corporation.

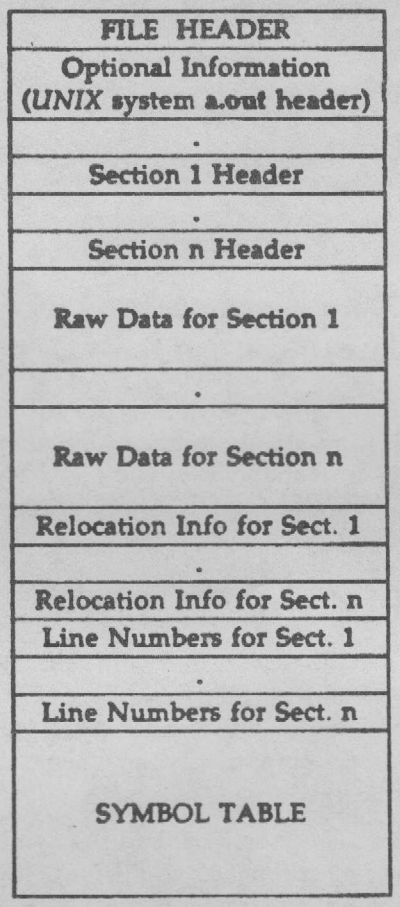


Figure 1. Object File Format

Section A section is the smallest portion of an object file that can be relocated and can be treated as one separate, distinct entity. In the default case, there are three sections, named **.text**, **.data**, and **.bss**. Additional sections are used for accommodating multiple text or data segments, shared data segments, or user-specified sections.

Physical Address This is the physical location in memory where a section is loaded.

Virtual Address This is the offset of a section with respect to the beginning of its segment or region. All relocatable references in a section assume that section occupies the virtual address at execution time.

3. File Header

The file header contains the twenty bytes of information described in Table 1. The last two bytes are flags that are used by *ld*. The UNIX Reference manual page for FILEHDR(4) gives the exact C language structure for the file header.

TABLE 1. FILE HEADER CONTENTS

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	Magic number as defined by the symbol MAGIC in file a.out.h.
2-3	unsigned short	f_nscns	Number of section headers (equals the number of sections)
4-7	long int	f_timdat	Time and date stamp indicating when the file was created relative to the number of elapsed seconds since 00:00:00 GMT, January 1, 1970.
8 -11	long int	f_symptr	File pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags, see Table 2.

The size of optional header information, f_opthdr, should be used by all referencing programs that need to seek to the beginning of the section header table. The size of the UNIX system optional header is currently 28 bytes on the VAX systems.

3.1 Flags

The last two bytes of the file header are flags that describe the type of the object file. The UNIX operating system version of the COFF has no use for some of these flags, but keeps them to maintain commonality.

TABLE 2. FILE HEADER FLAGS

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from file
F_EXEC	00002	File is executable (i.e. no unresolved external references)
F_LNNO	00004	Line numbers stripped from file
F_LSYMS	00010	Local symbols stripped from file
F_MINMAL	00020	Not applicable to UNIX operating system
F_UPDATE	00040	Not applicable to UNIX operating system
F_SWABD	00100	This file has had its bytes swabbed (i.e., the bytes of symbol table name entries have been reversed.)
F_AR16WR	00200	Created on AR16WR machine (e.g. PDP 11/70)
F_AR32WR	00400	Created on AR32WR machine (e.g. VAX-11/780)
F_AR32W	01000	Created on AR32W machine
F_PATCH	02000	Not applicable to UNIX operating system

The notation "AR16WR" in Table 2 signifies the architecture of the host machine where the file was created. The AR stands for architecture, the digits give the number of bits per word, W signifies left-to-right byte-ordering (most significant bit first), and WR signifies right-to-left byte-ordering (least significant bit first).

4. Optional Header Information

The template for optional information varies among different systems that use the COFF. Applications place all systems-dependent information into this record. General utility programs (e.g., the table access library functions, the disassembler, etc.) can be made to work properly on any Common object file by seeking past this record using the size of optional header information in the file header (bytes 16-17).

4.1 Standard *UNIX* System a.out Header

By default, files produced by the link editor always have a standard *UNIX* system a.out header in the optional header field. The VAX version of the optional header is 28 bytes long.

The fields of the optional header are described in table 3.

TABLE 3. OPTIONAL HEADER CONTENTS

Bytes	Declaration	Name	Description
0-1	short	magic	Magic number
2-3	short	vstamp	Version stamp
4-7	long int	tsize	Size of text
8 -11	long int	dsize	Size of initialized data
12-15	long int	bsize	Size of uninitialized data
16-19	long int	dum1	Unused dummy field
20-23	long int	dum2	Unused dummy field
24-27	long int	entry	Entry point
27-31	long int	text_start	Base address of text
32-35	long int	data_start	Base address of data

The magic number for the UNIX operating system is a machine-dependent constant that can be found in the header file a.out.h(4).

4.2 C Data Structure

The following C language *struct* declaration is currently used for standard *UNIX* system a.out file header:

```
typedef struct aouthdr {
        short   magic;          /* magic number */
        short   vstamp;         /* version stamp */
        long    tsize;          /* text size in bytes, padded */
                                /* to full word boundary */
        long    dsize;          /* initialized data size */
        long    bsize;          /* uninitialized data size */
#if     u3b
        long    dum1;           /* unused dummy field */
        long    dum2;           /* unused dummy field */
#endif
        long    entry;          /* entry point */
        long    text_start;     /* base of text for this file */
        long    data_start;     /* base of data for this file */
} AOUTHDR;
```

5. Section Headers

Every object file has a table of section headers to specify the layout of data within the file. Each section within an object file also has its own header.

The section header table consists of one entry for every section in the file. Each entry contains the following information about the section:

TABLE 4. SECTION HEADER CONTENTS

Bytes	Declaration	Name	Description
0-7	char	s_name	8 char null padded section name
8 -11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File ptr to relocation entries
28-31	long int	s_lnnoptr	File ptr to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	flags	Flags (see below)

Section sizes are always padded to a multiple of 4 bytes.

File pointers are byte offsets that can be used to directly locate the start of data, relocation, or line number entries for the section. They can be readily used with the *UNIX* system function fseek(3S).

5.1 Flags

The lower 4 bits of the flag field indicate a section type, defined as follows:

Mnemonic	Flag	Meaning
STYP_REG	0x00	regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	grouped section (formed from input sections)
STYP_PAD	0x08	padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	section contains only executable text
STYP_DATA	0x40	section contains only initialized data
STYP_BSS	0x80	section contains only unintialized data

The C language data structure that is used to declare section headers can be found on the UNIX Reference manual page SCNHDR(4).

5.2 .bss Section Header

The one anomaly in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size, symbols that refer to it, and symbols that are defined in it. At the same time a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are zero.

6. Sections

Figure 1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a full word boundary in the file.

Files produced by the UNIX system *cc* compiler and the *as* assembler always contain three sections, called **.text**, **.data**, and **.bss**. The **.text** section contains the instruction text (i.e., executable code), the **.data** section contains initialized data variables, and the **.bss** section contains uninitialized data variables.

The link editor "SECTIONS directives" (see **Link Editor User's Manual**) allow users to describe how input sections are to be combined, to direct where to place output sections, and to rename output sections. If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the compiler are linked together, each containing the three sections **.text**, **.data**, and **.bss**, the output object file will also contain three sections, **.text**, **.data**, and **.bss**.

7. Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the following 10-byte format:

TABLE 5. RELOCATION SECTION CONTENTS

Bytes	Declaration	Name	Description
0-3	long int	r_vaddr	(Virtual) address of reference
4-7	long int	r_symndx	Symbol table index
8-9	unsigned short	r_type	Relocation Type (see below)

The first 4 bytes of the entry is the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates what type of relocation is to be applied.

7.1 C Data Structure

The following structure declaration is currently used for relocation entries:

```
struct reloc {
        long    r_vaddr;        /* virtual address of reference */
        long    r_symndx;       /* index into symbol table */
        unsigned short          r_type;                 /* relocation type */
        };
```

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated.

Relocation types currently recognized are:

VAX RELOCATION TYPES		
Mnemonic	**Flag**	**Meaning**
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_RELBYTE	017	Direct 8-bit reference to the symbol's virtual address.
R_RELWORD	020	Direct 16-bit reference to the symbol's virtual address.
R_RELLONG	021	Direct 32-bit reference to the symbol's virtual address.
R_PCRBYTE	022	A "PC-relative" 8-bit reference to the symbol's virtual address.
R_PCRWORD	023	A "PC-relative" 16-bit reference to the symbol's virtual address.
R_PCRLONG	024	A "PC-relative" 32-bit reference to the symbol's virtual address.

On the VAX processors relocation of a symbol index of -1 indicates that the relative difference between the current segment's start address and the program's load address is added to the relocatable address.

The UNIX system *as* assembler automatically generates relocation entries which are then utilized by the link editor. The link editor uses this information to resolve external references in the file.

8. Line Numbers

When invoked with the proper option the UNIX system *cc* compiler generates an entry in the object file for every C language source line where a breakpoint can be inserted. Users can then reference line numbers when using a software debugger like sdb. All line numbers in a section are grouped by function, as shown below.

symbol index	0
physical address	line number
physical address	line number
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number zero, and has an index into the symbol table for the entry containing the function name in place of the physical address. Subsequent entries will have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries appear in increasing order of address.

8.1 C Data Structure

The following structure declaration is currently used for line number entries:

```
struct lineno
{
        union
        {
                long  l_symndx;          /* symtbl index of func name */
                long  l_paddr;           /* paddr of line number */
        } l_addr;
        unsigned short l_lnno;           /* line number */
};
```

9. Symbol Table

Because of symbolic debugging requirements for SGS the order of symbols in the symbol table is very important. Symbols appear in the sequence shown as below:

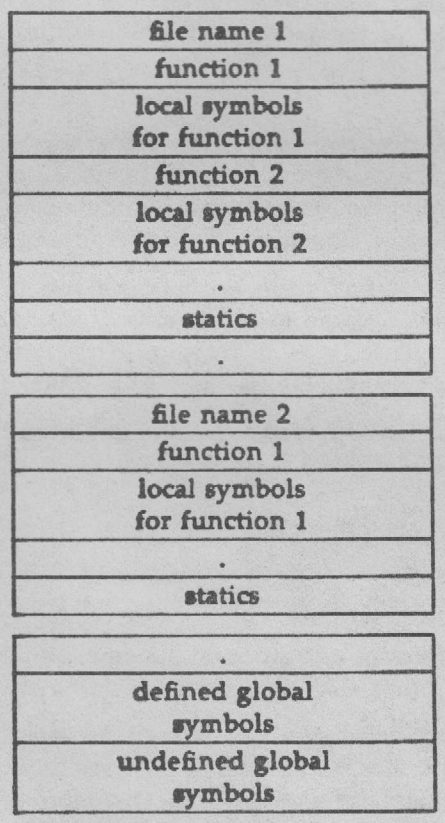


Figure 2. COFF Symbol Table

The word "statics" in Figure 2 means symbols defined in the C language storage class *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol, with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the name (null-padded), the structure value, and other information.

9.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler and link editor. These symbols are:

TABLE 6. SPECIAL SYMBOLS IN THE SYMBOL TABLE

Symbol	Meaning
.file	file name
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the function returned structure or union See "Symbol Table Entries," Section 1.8.2.
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration

Six of these special symbols occur in pairs. The .bb and .eb symbols indicate the boundaries of inner blocks; a .bf and .ef pair brackets each function; and a .xfake and .eos pair names and defines the limit of structures, unions, and enumerations that were not named. The .eos symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name (a perfectly legitimate C language syntax) the SGS must invent a name to be used in the symbol table. The name chosen for the symbol table is .xfake, where "x" is an integer. If there are 3 unnamed structures, unions, or enumerations in the source, their tag names will be ".0fake", ".1fake", and ".2fake".

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry. (See Sections 1.8.2 and 1.8.3.)

Inner Blocks

The C language defines a *block* as a compound statement that begins and ends with braces ({ and }). An *inner block* is a block that occurs within statements such as *if*, *switch*, or *while*.

For each inner block that has local symbols defined, a special symbol .bb is put in the symbol table immediately before the first local symbol of that block. Also a special symbol .eb is put in the symbol table immediately after the last local symbol of that block. The sequence is:

.bb
local symbols for that block
.eb

Because inner blocks can be nested by several levels, nested inner blocks may have the following sequence:

.bb for block 1
local symbols for block 1
.bb for block 2
local symbols for block 2
.eb for block 2
.
.bb for block m
local symbols for block m
.bb for block m1
local symbol for block m1
.
.bb for block mn
local symbols for block mn
.eb for block mn
.
.eb for block m1
.eb for block m
.
.eb for block 1

Symbols for Functions

For each function, a special symbol .bf is put between the function name and the first local symbol of the function in the symbol table. Also a special symbol .ef is put immediately after the last local symbol of the function in the symbol table. The sequence is:

function name
.bf
local symbols
.ef

If the return value of the function is a structure or union, a special symbol .target is put between the function name and the .bf. The sequence becomes:

function name
.target
.bf
local symbols
.ef

The UNIX system *cc* compiler invents .target to store the function-returned structure or union. The symbol .target is an automatic variable with "pointer" type. Its value field in the symbol table entry is always zero.

9.2 Symbol Table Entries

All symbols, regardless of storage class and type, have the same *format* for their entries in the symbol table. The symbol table entries each contain the following 18 bytes of information. It should be noted that indices for symbol table entries begin at zero and count upward.

TABLE 7. SYMBOL TABLE ENTRY FORMAT

Bytes	Declaration	Name	Description
0-7	char	n_name	8 char null-padded symbol name
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnum	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	storage class of symbol
17	char	n_numaux	number of auxiliary entries

The meaning of the entries in each of these fields is described below.

Symbol Names

The name of a symbol is currently limited to 8 characters; longer names are truncated by the *cc* compiler. Some special symbols are generated by the compiler and link editor, as discussed under **Special Symbols**. The names of special symbols always start with a dot ("."), e.g., **.file**, **.5fake**, **.bb**.

Storage Classes

The storage class field has one of the following values:

Mnemonic	Value	Storage Class
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_LABEL	6	label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter

Mnemonic	Value	Storage Class
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_ALIAS	105	duplicated tag

The SGS compress utility, cprs, generates the C_ALIAS mnemonic. This utility, which is described in the UNIX Reference manual, removes duplicated structure, union, and enumeration definitions and puts ALIAS entries in their places.

There are also some "dummy" storage classes defined in the header file. They are used only internally by the compiler and the assembler. These storage classes are:

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of function
C_EXTDEF	5	external definition
C_ULABEL	7	undefined label
C_USTATIC	14	uninitialized static
C_LINE	104	used only by utility programs

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are:

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Also some storage classes are used only for certain special symbols. They are summarized as follows:

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Symbol Value Field

The meaning of the "value" of a symbol depends on its storage class. This relationship is summarized as follows:

Storage Class	Meaning
C_AUTO	stack offset
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset
C_ARG	stack offset
C_STRTAG	0
C_MOU	offset
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see below)
C_ALIAS	tag index

If the current symbol is the last symbol in this object file that has storage class C_FILE (.file symbol), its value is the symbol table entry index of the first global symbol. Otherwise the symbol value equals the symbol table entry index of the next .file symbol. That is, the .file entries form a one-way linked list in the symbol table.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers have the following meaning:

Mnemonic	Section Number	Meaning
N_DEBUG	-2	special symbolic debugging symbol
N_ABS	-1	absolute symbol
N_UNDEF	0	undefined external symbol
N_SCNUM	1-077767	section number where symbol was defined

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value, but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and .eos symbols. A section number of zero indicates a relocatable external symbol that is not

defined in the current file. The .text, .data, and .bss symbols default to section numbers 1, 2, and 3 respectively.

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized as follows:

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. Each symbol has exactly one basic, or fundamental, type, but can have more than one derived type. The format of the 16-bit type entry is:

d6	d5	d4	d3	d2	d1	typ

Bits 0-3, called "typ", indicate one of the following fundamental types:

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point

T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Bits 4-15 are arranged as six 2-bit fields marked "d1" through "d6." These "d" fields represent levels of the following derived types:

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type.

*char *func();*

Here *func* is the name of a function that returns a pointer to a character. The fundamental type of *func* is 2 (character), the d1 field is 2 (function), and the d2 field is 1 (pointer). Therefore the type word in the symbol table for *func* would contain the hexadecimal number 0x62, which is interpreted to mean "function that returns a pointer to a character."

*short *tabptr[10][25][3];*

Here *tabptr* is a three dimensional array of pointers to short integers. The fundamental type of *tabptr* is 3 (short integer); the d1, d2, and d3 fields each contains a 3 (array), and the d4 field is 1 (pointer). Therefore the type entry in the symbol table would contain the hexadecimal number 0x7f3, indicating a "three dimensional array of pointers to short integers."

Type Entries and Storage Classes

The following table shows which type entries are legal for each storage class.

TABLE 8. TYPE ENTRIES BY STORAGE CLASS

Storage Class	----------"d" entry---------- Function?	Array?	Pointer?	"typ" entry basic type
C_AUTO	no	yes	yes	Note 1
C_EXT	yes	yes	yes	Note 1
C_STAT	yes	yes	yes	Note 1
C_REG	no	no	yes	Note 1
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Note 1
C_ARG	no	no	yes	Note 1
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Note 1
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Note 1
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Note 1
C_FIELD	no	no	no	Note 2
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	Note 3

Note 1: any except T_MOE Note 2: T_ENUM, T_UCHAR, T_USHORT, T_UINT, T_ULONG
Note 3: T_STRUCT, T_UNION, T_ENUM

Conditions for the "d" entries apply to d1 through d6, except that it is impossible to have two consecutive derived types of "function."

Although function arguments can be declared as arrays, they will be changed to pointers by default. Therefore no function argument can have "array" as its derived type.

Structure for Symbol Table Entries

The following C language structure declaration is currently used for symbol table entries:

```
struct syment
{
        char            n_name[SYMNMLEN];           /* symbol name */
        long            n_value;                    /* value of symbol */
        short           n_scnum;                    /* section number */
        unsigned short  n_type;             /* type and derived type */
        char            n_sclass;                   /* storage class */
        char            n_numaux;                   /* number of aux entries */
};

#define SYMNMLEN        8
#define SYMESZ          18                  /* size of a symbol table entry */
```

9.3 Auxiliary Table Entries

Currently, there is at most one auxiliary entry per symbol. The auxiliary table entry contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized as follows:

TABLE 9. AUXILIARY SYMBOL TABLE ENTRIES

Name	Storage Class	Type Entry d1	Type Entry typ	Auxiliary Entry Format
.file	C_FILE	DT_NON	T_NULL	file name
.text,.data,.bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG, C_UNTAG,C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
fcname	C_EXT,C_STAT	DT_FCN	Note 1	function
arrname	Note 2	DT_ARY	Note 1	array
.bb	C_BLOCK	DT_NON	T_NULL	beginning of block
.eb	C_BLOCK	DT_NON	T_NULL	end of block
.bf,.ef	C_FNC	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	Note 2	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Note 1: Any except T_MOE.

Note 2: C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In the above table "tagname" means any symbol name including the special symbol .xfake, and "fcname" and "arrname" represent any symbol name.

Any symbol that satisfies more than one condition in Table 7 should have a union format in its auxiliary entry. Symbols that do not satisfy any of the above conditions should NOT have any auxiliary entry.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are zero, regardless of the size of the entry.

Sections

The auxiliary table entries for sections have the format:

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	-	dummy	0

Tag Names

The auxiliary table entries for tag names have the format:

Bytes	Declaration	Name	Description
0-5	-	dummy	0
6-7	unsigned short	x_size	size of struct, union or enumeration
8-11	-	dummy	0
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	dummy	0

End of Structures

The auxiliary table entries for the end of structures have the format:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	dummy	0
6-7	unsigned short	x_size	size of struct, union or enumeration
8-17	-	dummy	0

Functions

The auxiliary table entries for functions have the format:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this function
16-17	unsigned short	x_tvndx	not used in UNIX system

Arrays

The auxiliary table entries for arrays have the format:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_lnno	line number of declaration
6-7	unsigned short	x_size	size of the array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	dummy	0

End of Blocks and Beginning and End of Functions

The auxiliary table entries for the end of blocks and the beginning and end of functions have the format:

Bytes	Declaration	Name	Description
0-3	-	dummy	0
4-5	unsigned short	x_lnno	C-source line number
6-17	-	dummy	0

Beginning of Blocks

The auxiliary table entries for the beginning of blocks have the format:

Bytes	Declaration	Name	Description
0-3	-	dummy	0
4-5	unsigned short	x_lnno	C-source line number
6-11	-	dummy	0

12-15	long int	x_endndx	index of next entry past this block
16-17	-	dummy	0

Names related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumerations symbols have the format:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	dummy	0
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	-	dummy	0

Names defined by "typedef" may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people {
        char name[20];
        long id;
        };

typedef struct people EMPLOYEE;
```

The symbol "EMPLOYEE" will have an auxiliary table entry in the symbol table but symbol "STUDENT" will not.

C Data Structure

The exact C language data structure for auxiliary symbol table entries can be found on the UNIX Reference manual page for SYMS(4).

10. Access Routines

Supplied with every standard UNIX system release are set of access routines that are used for reading the various parts of a Common Object File. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file. In this way, users can concern themselves with the section they are interested in without needed to know all the object file details.

The access routines can be divided into four categories:

1. Functions that open or close an object file
2. Functions that read header or symbol table information

3. Functions that position an object file at the start of a particular section of the object file
4. A function that returns the symbol table index for a particular symbol

These routines can be found in /lib/libld.a and are listed in section 3 of the UNIX system Reference manual. A summary of what is available can be found in the UNIX system Reference manual under LDFCN(4).

User Level Changes to the C Language

User Level Changes to the C Language

Bell Laboratories

1. Introduction.

This document describes the latest release of the Portable C Compiler (PCC). PCC represents the machine independent portion of the C compiler found on UNIX* systems. This release is identified as PCCr1.1. It encompasses revisions to the C language, as described in [1], bug fixes and various implementor enhancements over the last release.

2. C Language Changes.

2.1 Addition of Enumeration Data type.

Enumerations are unique data types with named constants. They replace, in part, the use of *#define* constants, and offer the advantage of scoped constant names and strong type checking in the use of such names. As an example:

```
enum color {red, green, yellow, blue};

...

enum color *cp, col;

...

col = yellow;
cp = &col;

...

if ( *cp == green) ...
```

This example declares the identifier "color" to be the enumeration-tag of a type describing various colors, and then declares "cp" as a pointer to an object of that type and "col" as an object of that type. The role of the identifier is entirely analogous to that of the structure-tag in a struct-specifier; it names a particular enumeration.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types. The compiler maps enumerations into the **int** storage class.

A complete, formal specification can be found in section 8.5 of [1].

2.2 Structure and Union Assignment.

Structure and union assignment was added to the C language to simplify both the source and object code used to transfer the value of one structure or union instance to another and to allow functions

---

* UNIX is a trademark of Bell Telephone Laboratories, Inc.

to return aggregate values. Structure and union assignment also improves source program readability.

Structures and unions may be assigned as a unit, passed as arguments to functions, or returned by functions. The types of all structure operands taking part in these operations must be the same.

An example is as follows:

```
struct point {
        int     x, y;
} val;

...

extern struct point func();

...

val = func();

...
```

This example will assign the structure returned by function "func" to the structure instance "val".

2.3 Nonunique Structure Member Names.

In previous language specifications all symbol names, whether simple variables, structure and union members or typedef names, had to be unique throughout the program. The C language has been revised to permit the reuse of structure members or field names, except that a particular symbol may not be used for two distinct names in the same structure. These nonunique member names permit more natural structure and union member naming conventions and result in stronger type checking of both structure and union member references.

This change should not cause any upward compatibility problems, as the following example will indicate:

```
struct {
        short   count;
        int     sum;
} a;

...

struct {
        char    *ptr;
        long    count;
} b;
```

In previous compiler releases the above example would have been flagged as an error since the variable "count" was a declared once as short and then redeclared as a long. even though the declarations appeared in different structures. With the current release the nonunique variable name is allowed and must be uniquely described through the use of complete structure name qualifiers.

2.4 Complete Structure Qualifications are Required.

A structure (union) member reference is a chain of member references, also known as qualifications, that are prefixed by a structure (union) or a pointer to one. Each qualification implies the addition of an offset within an address computation. Previous compilers allowed the omission of those qualifications with an offset of zero. That is, they failed to check if the variable was a member of the appropriate structure (union).

An example of this is:

```
struct {
        union {
                long    l;
                short   s[2];
        } word;
        float   f;
} x;

ptr = x.l;
ptr = x.s[0];
```

Prior to the language change these references were considered legitimate. However with the current release of PCC these will produce non-fatal error messages. The proper usage in the above example is:

```
ptr = x.word.l;
ptr = x.word.s[0];
```

2.5 Addition of Type Specifier Void.

This release of the compiler introduces a new type specifier, **void.** The **void** type specifies an empty set of values. It should be used as the type returned by functions that generate no return value.

An example of its use is as follows:

```
void error(str)
char    *str;
{
        printf("Error: %s\n", str);
}

...

(void) error("missing variable");

...
```

In this example there is no value returned from the function error. This is explicitly specified by the use of the void specifier on the function calling line.

Functions that have no return type defined act the same as in the past, that is they default to int.

2.6 Old Style Initialization Removed.

The old style of variable initialization is no longer accepted by the compiler. Any code that relies on this feature will produce a fatal diagnostic when recompiled with the latest compiler. A common class of user syntax errors caused the compiler to incorrectly generate initialization code with undesirable effects (e.g., core dumps) and inappropriate diagnostics.

An example of such initialization follows:

```
long    x  1;
int     f  (2);
```

The correct syntax for this example is:

```
long    x = 1;
int     f = (2);
```

2.7 Old Style Assignment Operators Removed.

Old style assignment operators have been removed from the C language. This release will flag their use as a fatal error and produce an appropriate message. Earlier versions of C allowed the form *=op* instead of *op=* for assignment operators.

Examples of such assignment operations are:

```
a =+5;
x =-1;
```

The "=+" is now a syntax error and will be flagged by the compiler. The compiler also flags the "=–" due to its ambiguity with the assignment of a negative value. In the old semantics this operation would imply that a 1 should be subtracted from the variable x. However, in the next release of the compiler the interpretation will be to assign the constant (–1) to the variable x.

2.8 Octal 8 and 9 Produce Warning Message.

The use of digits 8 and 9 as octal numbers is no longer part of the language. In previous releases the compiler accepted the digits 8 and 9 in octal constants. As of this implementation 8 and 9 will be flagged with a warning message when they appear in octal constants. The compiler will convert the appearance of an 8 or 9 into an octal 010 or 011, respectively. Arithmetic will be performed to add the carry to the next significant digit. Future releases will make such usage a fatal error.

An example of this is:

```
x = 029;
```

The compiler will produce a warning and treat the constant as if it was (020 + 011) = 031.

3. Bug Fixes.

Various enhancements and bug fixes were done to the compiler since System III. Numerous modification requests revealed many bugs that were identified and corrected. This document will not attempt to address every change that occurred since the last release but merely highlight the ones that are most likely to be observed.

3.1 Pointer to Function Returning Void.

Additional type checking code has been added to the compiler to allow pointer to a function that returns void. A declaration of such is:

```
void (*f)();
```

3.2 Access to a Member of a Returned Structure or Union.

Now that the language allows structure and union assignments a bug was uncovered in the compiler regarding the immediate access of members of the returned structure or union.

An example of such a construct is:

```
x = f().a;
```

where f() is a function returning a structure that has a member element of "a".

This bug has been identified and removed from the compiler.

3.3 Conversion Code for Assignment Operators.

PCC now produces the correct type conversion code for assignment operators embedded in expressions. All questionable code which relied upon incorrect boundary handling should be re-examined. A common example of such incorrect coding is shown below:

```
char ch;

while ( ( ch = getchar() ) != EOF )

...
```

The EOF is defined to be a negative integer constant while the expression on the left of the " != " is an unsigned character. The EOF returned by getchar() has been converted from an integer to a character value and no longer acts like the original EOF. To correct this example, the variable "ch" should be declared as an integer.

3.4 Parameters in Function Declarations.

A serious bug was uncovered when parameters were inadvertently placed into function declarations. An example of such is:

```
short    func(x, y);
```

Here the user is simply declaring the function "func" to return a short, not actually defining the body of the function. In this case the variable arguments "x" and "y" represent syntax errors. The compiler now checks for this and reports it as a fatal error.

3.5 Constants.

3.5.1 The tests to determine when constants become long integers have been changed to conform to the C Reference Manual. A problem existed with these tests in a cross-compiling environment where the size of int and long were different on the host and target machines.

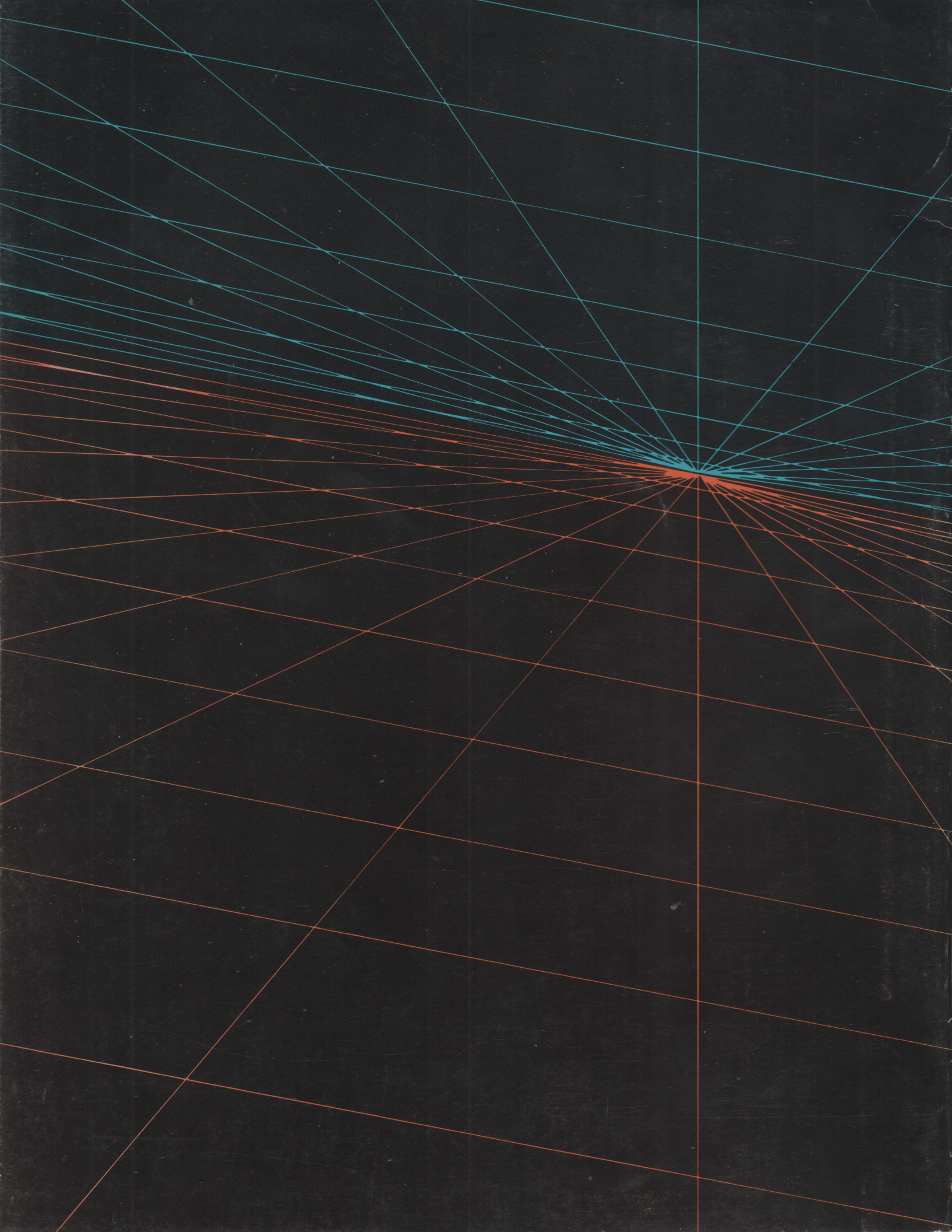
3.5.2 Constant folding of long constants has been fixed. In some constant expressions involving long constants the correct type (long) was not given to the result.

3.5.3 Constant folding of unsigned constants has been fixed. The latest compiler implemented the language specification that **sizeof** returns an unsigned quantity. Since there was no knowledge of unsigned constants in the routine that does constant folding, the unary operators negate ("–") and complement ("~") sometimes produced the wrong results when applied to the value returned by **sizeof**.

3.5.4 Illegal hexadecimal constants are no longer allowed. Previous releases of the compiler had a bug in the code that checked for hex constants. The bug would allow such expressions as:

```
i = 010x;
```

as a valid hex constant when the value 0x10 was intended.

REFERENCES

[1] Ritchie, D. M. "The C Programming Language-Reference Manual", September 1980.